Enterprise Architektur-Muster

JULIAN BRUDER*, ABDELLAH FILALI*, and LUCA FRANKE*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

Blah …

1 EINLEITUNG

Blah …

2 GRUNDLAGEN VON ENTERPRISE-ARCHITEKTUREN

Blah …

3 KLASSISCHE ENTERPRISE-ARCHITEKTUREN

3.1 Monolith

Der Begriff *Monolith* stammt aus dem Altgriechischen und bedeutet *einheitlicher Stein*. Die monolithische Architektur beschreibt ein Softwarearchitektur-Pattern, die die gesamte Funktionalität in einer einzigen Anwendung bündelt, wobei ein einzigen Prozess für die Ausführung der Anwendung zuständig ist [Chen et al. 2017, 1].

Anwendungen dieser Architektur bestehen aus eng gekoppelte Komponenten, die von einander abhängig sind, sodass sie weder eigenständig laufen noch in manchen Fällen nicht isoliert kompiliert werden können. [Salaheddin and Ahmed 2022]

Abb. 1. Monolith Architektur

Diese Architektur weist für kleinere Anwendungen gewisse Vorteile auf, wie eine einfache testbarkeit, Logging, Deployment, sowie Debugging. Zudem eine Datenbanksynchronisation ist nicht notwendig, da die gesamte Daten in einer Datenbank persistiert werden. [?, 2]

Betrachten wir das E-Commerce-Beispiel. Dafür definieren wir drei Klassen:

- `OrderService`: Klasse, die die Bestellungen verwaltet
- `PaymentService`: Klasse, die die Zahlungen abwickelt
- `ShipmentService`: Klasse, die die Lieferungen initiiert

Die Kommunikation zwischen den Klassen erfolgt durch Methodenaufrufe. Dabei ist die Klasse `OrderService` die Hauptklasse, die die anderen Klassen verwendet, um den Bestellungsvorgang durchzuführen. Vorteilhaft hier ist, dass hier die Kommunikation zwischen den komponenten einfach ist. Durch die Verwendung von Methodenaufrufen wird die Komplexität reduziert, die mit Intersystemkommunikationen verbunden ist.

Jedoch, wenn die Code-Basis wächst und komplex wird, treten einige Nachteile auf. Durch Änderungen in einer Komponente können unerwartete kaskadierende Fehler auslösen, was die Weiterentwicklung in kleinen autonomen Teams erschwert und verlangsamt.

Außerdem durch die Verwendung von direkte Methodenaufrufe, entsteht eine enge Kopplung zwischen den Komponenten. Änderungen in der Implementierung oder Signatur einer Methode können zu kaskadierenden Fehlern führen.

Weiterhin ist die Wiederverwendung von Funktionalitäten nicht möglich, da die Komponenten eng gekoppelt sind. Das führt zu Duplikation von Code, was die Änderungen teurer macht, da die Änderungen in mehreren Stellen durchgeführt werden müssen.

Die dadurch erhöhte Komplexität und die schwierige Wartung führen zu längere Iterationen. Da das Deployment erfolgt nur als Ganzes und die Iterationen länger dauern, kommt es zu seltenen Auslieferungen von neuen Features. Ferner ist Horizontale Skalierung auch nicht möglich, da die Anwendung nur als Ganzes skaliert werden kann. Insgesamt ist eine gute Einstiegslösung, allerdings nicht für größere Anwendungen geeignet, da die Agilität davon stark betroffen wird.

3.2 Modular Monolith

Das Hauptproblem der vorherigen Architektur ist die eng gekoppelte Natur der Architektur, die eine gewisse Komplexität und Unflexibilität mit sich bringt. Modulare Monolithen sind eine Evolution der Monolithic Architektur, die die Vorteile der Monolithic Architecture erben und gleichzeitig die Nachteile der enge Kopplung verringern.

In diese Architektur wird die Code-Basis in mehrere Module geteilt, die jeweils eine Teil-Funktionalität der Anwendung implementieren. Die Kommunikation zwischen den Module erfolgt durch klar definierte Interfaces, wobei die Wiederverwendbarkeit und Austauschbarkeit der Module ermöglicht wird.[Su and Li 2024, 11]

Die Trennung in Modulen reduziert zudem die Komplexität und fördert eine bessere Organization der Code-Basis, was zu einer bessere Wartbarkeit führt.[Barde 2023, 2]

Abb. 2. Modular Monolith Architektur

Betrachten wir erneut das E-Commerce-Beispiel. Diesmal wird die Anwendung in drei Hauptmodule aufgeteilt:

- `OrderModule`: Verantwortlich für die Verwaltung der Bestellungen
- `PaymentModule`: Verantwortlich für die Abwicklung der Zahlungen
- `ShipmentModule`: Verantwortlich für die Initiierung der Lieferungen

Abhängigkeiten zwischen den Modulen werden durch die Verwendungen von Interfaces reduziert, indem die Module nur die

*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt.

Schnittstellen kennen und nicht die Implementierung. Zum Beispiel Änderungen an der Implementierung einer Komponente von `PaymentModule` müssen nur in der Implementierung des Interfaces durchgeführt werden, ohne dass die anderen abhängigen Module davon betroffen werden. Die Modularisierung ermöglicht zusätzlich eine verbesserte Entwicklung in semi-autonomen Teams im Vergleich zu den traditionellen Monolithen. Problematisch ist jedoch, dass die Anwendung nur als Ganzes deployt werden kann, was die Iterationen verlangsamt. Außerdem ist weiterhin keine horizontale Skalierung möglich und die Funktionalitäten können nicht wiederverwendet werden, da diese immer noch Teil einer einzigen Anwendung und eng miteinander gekoppelt sind.

3.3 Layered

Das Layered Architektur-Pattern, auch bekannt als n-Tier-Architektur-Pattern, beschreibt eine Architektur, die die Anwendung in Schichten aufteilt.

Ein wichtige Eigenschaft der Layered Architecture ist der die Trennung der Zuständigkeiten (Englisch *Separation of concerns*). Komponenten mit unterschiedlichen Aufgaben sollten auf verschiedene Schichten verteilt werden, sodass die Komponenten einer Schicht jeweils für eine klar definierte, gemeinsame Aufgabe zuständig sind. [Tu 2023, S. 34]

Obwohl diese Architektur keine feste Anzahl an Schichten vorschreibt, bestehen die Layered Architectures meistens aus 3 Schichten [Tu 2023]:

- Presentations-Layer: Verantwortlich für die Interaktion mit dem Benutzer und die Weiterleitung von Benutzeranfragen an die Business-Layer.
- Business-Layer: Zuständig für die Verarbeitung von Geschäftslogik und Regeln, die Weiterleitung von Daten zwischen die Presentation-Layer und die Data-Layer.
- Data-Layer: Verantwortlich für die Interaktion mit der Datenbank und die Kommunikation mit der Business-Layer, um Daten bereitzustellen oder die Ergebnisse entweder an die Presentation-Layer oder zurück an den Datenspeicher zu übermitteln.

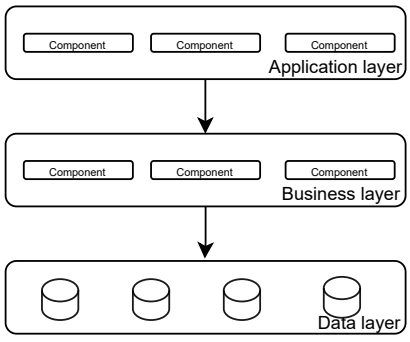

Abb. 3. Layered Architektur

Ein zentrales Prinzip der Layered Architecture ist der Isolation von Schichten besagt, dass die Kommunikation zwischen Schichten ausschließlich über definierte Schnittstellen erfolgen sollte. Dadurch wirken sich Änderungen in einer Schicht nur auf die Schicht selbst sowie höchstens auf eine andere Schicht aus. [Richards 2015, 10]

Ein weiteren relevantes Konzept ist die Unterscheidung zwischen offenen und geschlossenen Schichten. Dieses Konzept beschreibt, wie die Kommunikation zwischen den Schichten organisiert ist. Standardmäßig sind die Schichten auf Closed gesetzt. Das bedeutet, dass Module eine Schicht nur die Dienste der direkt darunterliegenden Schicht nutzen können. [Richards 2015][S. 3].(Siehe Abb. 4)

Diese Struktur fördert die Isolation von Schichten und reduziert somit die Abhängigkeiten zwischen den Schichten, da Schichten nur Abhängigkeiten zu den direkt darunterliegenden Schichten haben und nicht zu Schichten, die oberhalb liegen. Dadurch werden zyklischen Abhängigkeiten zwischen Schichten vermieden, was Änderungen in einer Schicht erleichtert.

Im Gegensatz dazu erlauben offene Schichten den Zugriff auf die Funktionalitäten von Schichten, die beliebig weit entfernt sind und nicht nur auf die Funktionalitäten der unmittelbaren darunterliegenden Schicht. (Siehe Abb. 4). [Richards 2015, 4]

Das kann besonders nützlich sein, wenn z.B. Anfragen kein Mehrwert durch das Durchlaufen eine bestimmte Schicht bwz. Schichten erhalten und nur weitergeleitet werden. Dieses Phänomen wird auch als `sinkhole anti-pattern` bezeichnet und das kann durch die Verwendung von offene Schichten vermieden werden.

Einerseits reduziert die Nutzung von offenen Schichten die unnötigen Aufwand für die Logik der Weiterleitung von Anfragen, andererseits verletzt das das Prinzip der Isolation von Schichten. Folglich Änderungen in einer Schicht können sich auf mehrere Schichten auswirken, was die Flexibilität und Wartbarkeit des Systems beeinträchtigt.[7 - 8][Richards 2015]

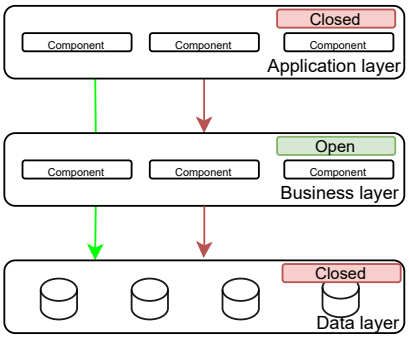

Abb. 4. Open/Close Layering

Die Implementierung dieser Architektur bietet eine Reihe von Vorteilen. Durch die Aufteilung des Systems in verschiedene Schichten wird eine unabhängige Entwicklung und Wartung der einzelnen Schichten ermöglicht [Tu 2023][S. 35]. Ein weiterer Vorteil besteht darin, dass der Zugriff auf die Dienste zwischen den Schichten über Schnittstellen (Interfaces) erfolgt. Dadurch sind Entwickler nicht gezwungen, die interne Implementierung einer Schicht zu kennen, um die bereitgestellten Dienste nutzen zu können. [Thomas and Webb 2024][S. 11] Aufgrund der losen Kopplung der Schichten ist es zudem möglich, neue Funktionalitäten hinzuzufügen durch die

Einführung neuer Schichten ohne dabei umfangreiche Anpassungen am gesamten System vornehmen zu müssen [Tu 2023][S. 35]. Allerdings führt die Nutzung dieser Architektur zu einer geringeren Performance, da für den Zugriff auf Dienste in unteren Schichten eine Kette von Anfrageweiterleitungen erforderlich ist [Thomas and Webb 2024][S. 11].

4 MODERNE ENTERPRISE-ARCHITEKTUREN

4.1 Event-Driven Architecture

Die Event-Driven Architecture wählt als Basis einen anderen Ausgangspunkt als die bisherigen Architekturmuster. Während bei letzteren Komponenten Dienste bereitstellen, welche von anderen Komponenten explizit genutzt werden, verhalten sich Dienstbereitstellende Komponenten in der Event-Driven Architecture reaktiv, werden also implizit von Dienst-konsumierenden Komponenten genutzt [Garlan and Shaw 1994]. Ein System reagiert somit asynchron auf Zustandsänderungen, also Ereignisse in diesem System [Manchana 2021]. Die in dieser Architektur minimalen Einheiten, welche Informationen einer Zustandsänderung kapseln, werden *Events* genannt. Die Idee der impliziten Behandlung von Ereignissen ist nicht neu und taucht erstmals 1994 im von Garlan und Shaw publizierten Papier „*An introduction to Software Architecture*“ auf.

Betrachten wir im Folgenden die Basis-Bestandteile der Event-Driven Architecture:

- Ereignis (englisch *Event*): Kapselt Information einer Zustandsänderung eines Systems
- Produzent (englisch *Producer*): Komponente, die Event erzeugt
- Herausgeber (englisch *Publisher*): Komponente, die, von Produzenten erzeugte, Events publiziert
- Konsument (englisch *Consumer*): Reagiert auf publizierte Events
- Vermittler (englisch *Mediator*): Liegt zwischen Produzenten und Konsumenten - filtert Events und verteilt diese auf Konsumenten
- Event-Bus: Oft auch *Event-Broker* genannt - bietet die Infrastruktur für die Gesamtheit der Vermittler

Abstrakt kann ein Event als ein Vertrag zwischen Produzenten und Konsumenten am Event-Bus betrachtet werden. Der Konsument nutzt die Spezifikation des Events am Bus, der Produzent implementiert jene Spezifikation. Abbildung 5 stellt diesen Vertrag dar.

Durch den Vertrag weisen die Events am Event-Bus starke Kohäsion und somit lose Kopplung auf. Diese lose Kopplung minimiert nicht nur kaskadierende Fehler, sondern ermöglicht agilen Entwickler-Teams durch klar abgegrenzte Features einfach definierbare Iterationen - eine Menge von Events, deren Erzeugung und Konsumierung.

Weiter sind Events oft nah an dem, was Ereignisse in realen Prozessen sind, also domain-driven. Gebündelt ermöglichen obige Punkte die kontinuierliche Auslieferung von Software in kurzen Intervallen.

Außerdem garantiert die asynchrone Behandlung von Ereignissen zusammen mit der loosen Kopplung maximale Skalierung. Daher sind Event-Driven Architekturen besonders für datenintensive

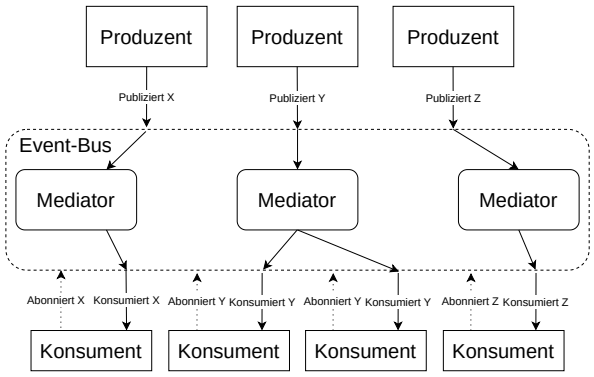

Abb. 5. Vertrag zwischen Produzenten und Konsumenten am Event-Bus

Echtzeit-Anwendungen wie IoT (Internet of Things) und Analytics geeignet [Siddiqui et al. 2023].

Die Agilität der Architektur kann weiter erhöht werden, indem der event-basierte Aspekt mit weiteren agilen Strukturen wie Microservices oder cloud-nativen Serverless-Functions kombiniert wird. Die damit einhergehende Komplexität stellt teilweise hohe Anforderungen an die Entwickler. Aufgrund der Asynchronität der Behandlung von Ereignissen ist die Testung des Systems meist schwer und die Fehlerbehandlung essentiell. Mögliche Problemquellen schließen dabei unter anderem Event-Verlust, erhöhte Latenz und Inkonsistenz ein. Die hohen Anforderungen an die Entwickler verlangen viel Vertrauen in jene, einer der zentralen Punkte des agilen Manifests [Michl 2018]. Insgesamt weist die Event-Driven Architecture also eine sehr hohe Agilität auf und ist damit besonders für moderne Software und ihre stetig wechselnden Anforderungen geeignet.

5 FALLSTUDIEN UND PRAXISBEISPIELE

Blah …

6 DISKUSSION

7 ZUSAMMENFASSUNG UND AUSBLICK

LITERATUR

Kalpesh Barde. 2023. Modular Monoliths: Revolutionizing Software Architecture for Efficient Payment Systems in Fintech. (2023). https://doi.org/10.1145/3643657.3643911

Rui Chen, Shanshan Li, and Zheng Li. 2017. From Monolith to Microservices: A Dataflow-Driven Approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 466–475. https://doi.org/10.1109/APSEC.2017.53

David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture.* Technical Report CMU/SEI-94-TR-021. https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/ Accessed: 2025-Jan-2.

Ramakrishna Manchana. 2021. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)* 10 (01 2021), 1706–1716. https://doi.org/10.21275/SR24820051042

Thomas Michl. 2018. *Das agile Manifest – eine Einführung*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–13. https://doi.org/10.1007/978-3-662-57699-1_1

Mark Richards. 2015. *Software architecture patterns.* Vol. 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA ….

Nada Salaheddin and Nuredin Ahmed. 2022. MICROSERVICES VS. MONOLITHIC ARCHITECTURE [THE DIFFERENTIAL STRUCTURE BETWEEN TWO ARCHITECTURES]. *MINAR International Journal of Applied Sciences and Technology* 4 (10 2022), 484–490. https://doi.org/10.47832/2717-8234.12.47

Hassaan Siddiqui, Ferhat Khendek, and Maria Toeroe. 2023. Microservices based architectures for IoT systems - State-of-the-art review. *Internet of Things* 23 (2023), 100854. https://doi.org/10.1016/j.iot.2023.100854

Ruoyu Su and Xiaozhou Li. 2024. Modular Monolith: Is This the Trend in Software Architecture?. In *Proceedings of the 1st International Workshop on New Trends in Software Architecture* (Lisbon, Portugal) *(SATrends '24)*. Association for Computing Machinery, New York, NY, USA, 10–13. https://doi.org/10.1145/3643657.3643911

Richard Thomas and Brae Webb. 2024. Layered architecture. (2 2024). https://csse6400.uqcloud.net/handouts/layered.pdf

Zhenan Tu. 2023. Research on the Application of Layered Architecture in Computer Software Development. *Journal of Computing and Electronic Information Management* 11 (11 2023), 34–38. https://doi.org/ojs/index.php/jceim/article/view/14398

A ANHANG 1

A.1 Übungsaufgaben

Blah …

B ANHANG 2

Blah …